



## Atomicity and Cache Coherency For x86

# Atomicity and Cache Coherency For SMP x86 Systems

Jim Vaught

March 2, 2007

[http://web.ics.purdue.edu/~jvaught/education/x86\\_memory.pdf](http://web.ics.purdue.edu/~jvaught/education/x86_memory.pdf)



# Atomicity and Cache Coherency For x86

## Overview of Lecture

- Cache Coherency
- Atomicity
- Implementation of Synchronization Primitives



## Cache Coherency: The MESI Protocol



State can be

- M (Modified)
- E (Exclusive)
- S (Shared)
- I (Invalid)



## Atomicity and Cache Coherency For x86

### MESI States

#### I - Invalid

- This line is invalid (empty)
- This processor does not own this line
- Another processor may have this line in M or E
- Other processors may have this line in S
- This line may be inconsistent with memory



## Atomicity and Cache Coherency For x86

### MESI States

#### S - Shared

- This line is valid
- This processor does not own this line
- No other processor has this line in M or E
- Other processors may have this line in S
- This line is consistent with memory



## Atomicity and Cache Coherency For x86

### MESI States

#### **E - Exclusive**

- This line is valid
- This processor owns this line
- No other processor has this line in M or E
- No other processor has this line in S
- This line is consistent with memory



## Atomicity and Cache Coherency For x86

### MESI States

#### M - Modified

- This line is valid
- This processor owns this line
- No other processor has this line in M or E
- No other processor has this line in S
- This line may be inconsistent with memory



# Atomicity and Cache Coherency For x86

## Example: Coherent System State

Memory:

A	0x12341234
---	------------

P0

S	TAG <sub>A</sub>	0x12341234
---	------------------	------------

P1

I	TAG <sub>A</sub>	0x11111111
---	------------------	------------

P2

--	--	--

P3

S	TAG <sub>A</sub>	0x12341234
---	------------------	------------



# Atomicity and Cache Coherency For x86

## Example: Incoherent System State

**INVALID!**

Memory:

A	0x12341234
---	------------

P0

M	TAG <sub>A</sub>	0x44444444
---	------------------	------------

P1

I	TAG <sub>A</sub>	0x12341234
---	------------------	------------

P2

S	TAG <sub>A</sub>	0x12341234
---	------------------	------------

P3

--	--	--



## Atomicity and Cache Coherency For x86

### Example: Coherent or Incoherent?

Memory:

A	0x12341234
---	------------

P0

S	TAG <sub>A</sub>	0x11111111
---	------------------	------------

P1

S	TAG <sub>A</sub>	0x11111111
---	------------------	------------

P2

S	TAG <sub>A</sub>	0x11111111
---	------------------	------------

P3

S	TAG <sub>A</sub>	0x11111111
---	------------------	------------



## Atomicity and Cache Coherency For x86

### Memory Requests

For WB memory type, requests **must** satisfy in cache

Request Types:

- load
- store
- load\_with\_store\_intent (aka RFO)
- load\_lock
- store\_unlock



## Atomicity and Cache Coherency For x86

### Memory Request: load

- Core needs to read memory location it may never write
- Cache line needs to be valid (S, E, or M)
- Example: compare instruction

```
CMP A, 11 { load tmp ← A  
           { sub tmp, 11  
           { set_flags
```



## Atomicity and Cache Coherency For x86

### Memory Request: store

- Core needs to write memory location it may not have read
- Cache line needs to be owned (E or M)
- Cache line will become M state
- Example: store LDTR instruction

SLDT A { store A←LDTR



## Atomicity and Cache Coherency For x86

### Memory Request: `load_with_store_intent`

- Core needs to read memory location it will soon write
- Cache line needs to be valid (S, E, or M)
- Bus unit should begin trying to take ownership of line
- Example: subtract instruction

```
SUB A, 11 { load_with_store_intent tmp←A  
           { sub tmp, 11  
           { store A←tmp  
           { set_flags
```



## Atomicity and Cache Coherency For x86

### Bus Transactions

Issued when memory request can't satisfy in cache

- BRL (Bus Read Line)  
Bring data into cache, but ownership not required
- BRIL (Bus Read Invalidate Line)  
Bring data into cache and take ownership
- BIL (Bus Invalidate Line)  
Take ownership of line already in the cache (no read)



## Atomicity and Cache Coherency For x86

### Bus Transactions For Memory Requests

	M	E	S	I
load				BRL
store		state=M	BIL	BRIL
LWSI			BIL	BRIL



## Atomicity and Cache Coherency For x86

### Snooping

- Each processor snoops the transactions of all others
- If snoop hits cacheline its state is updated accordingly
- In snoop phase of transaction processors give snoop response
- In data phase of transaction processor may drive data



## Atomicity and Cache Coherency For x86

### Snoop Types

GOTO\_I snoop (BRIL or BIL)

- State becomes I
- If S or E line hit, respond clean
- If M line hit, respond HITM

GOTO\_S snoop (BRL)

- If S or E line hit, respond HIT, state becomes S
- If M line hit, respond HITM, state becomes I



## Atomicity and Cache Coherency For x86

### Snoop Responses

Clean - No other processor has this line valid

- Processor brings line in in E state

HIT - Another processor has this line in S

- Processor brings line in in S state

HITM - Another processor had this line in M, now I

- Data driven by owning processor (implicit writeback)
- Chipset updates memory with HITM data
- Processor brings line in in E state



## Atomicity and Cache Coherency For x86

### Writebacks

- Process by which M state data is committed to system memory
- Implicit writeback (IWB) - M state line hit by snoop  
Data driven during transaction which caused the snoop
- Explicit writeback (EWB) - M state line capacity evicted  
Unsnopped transaction initiated by evicting processor



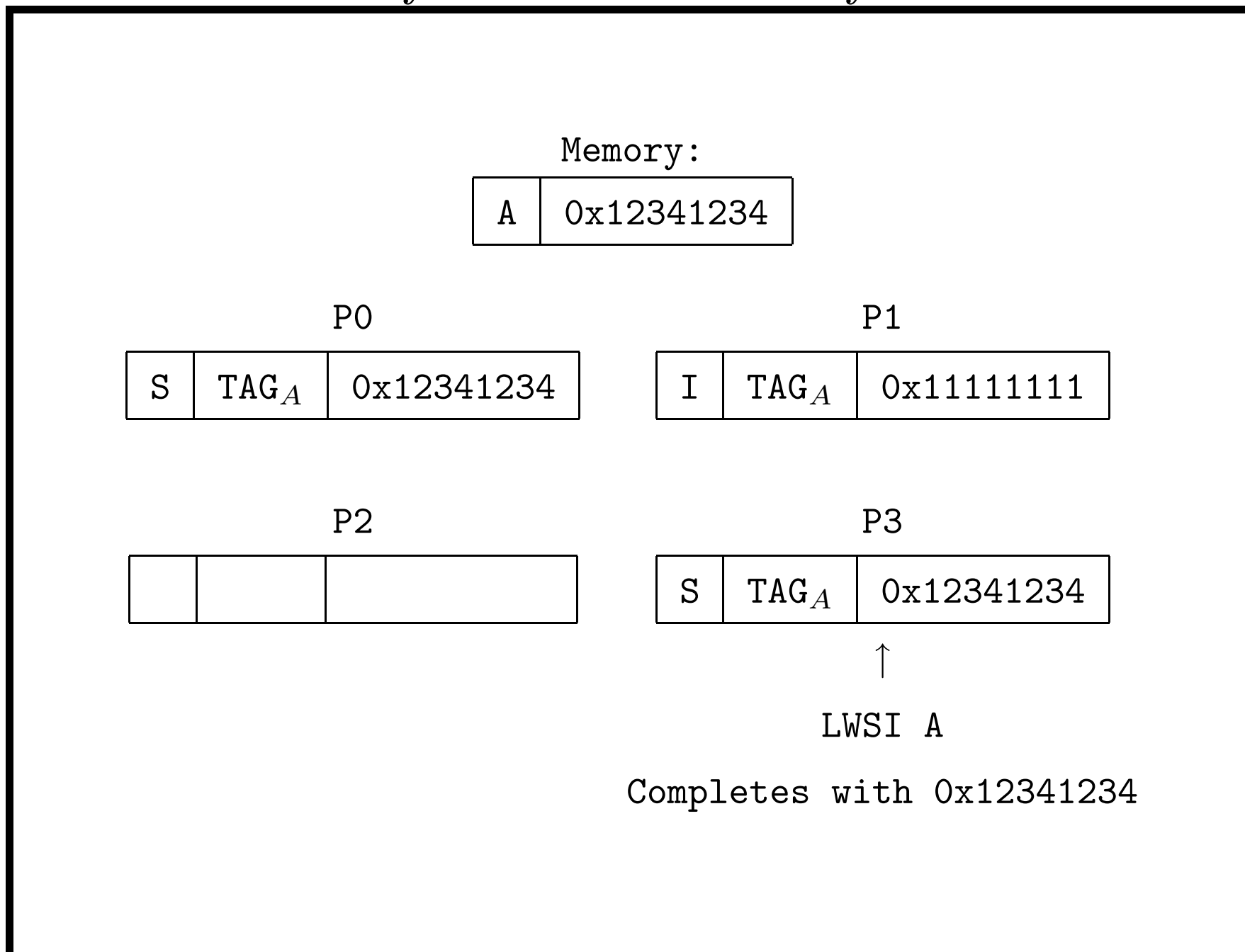
## Atomicity and Cache Coherency For x86

### Putting It All Together (Example)

- Start with first system state example
- P3 executes: SUB A, 0x11
- P0 executes: CMP A, 0x11
- P1 executes: CMP A, 0x11

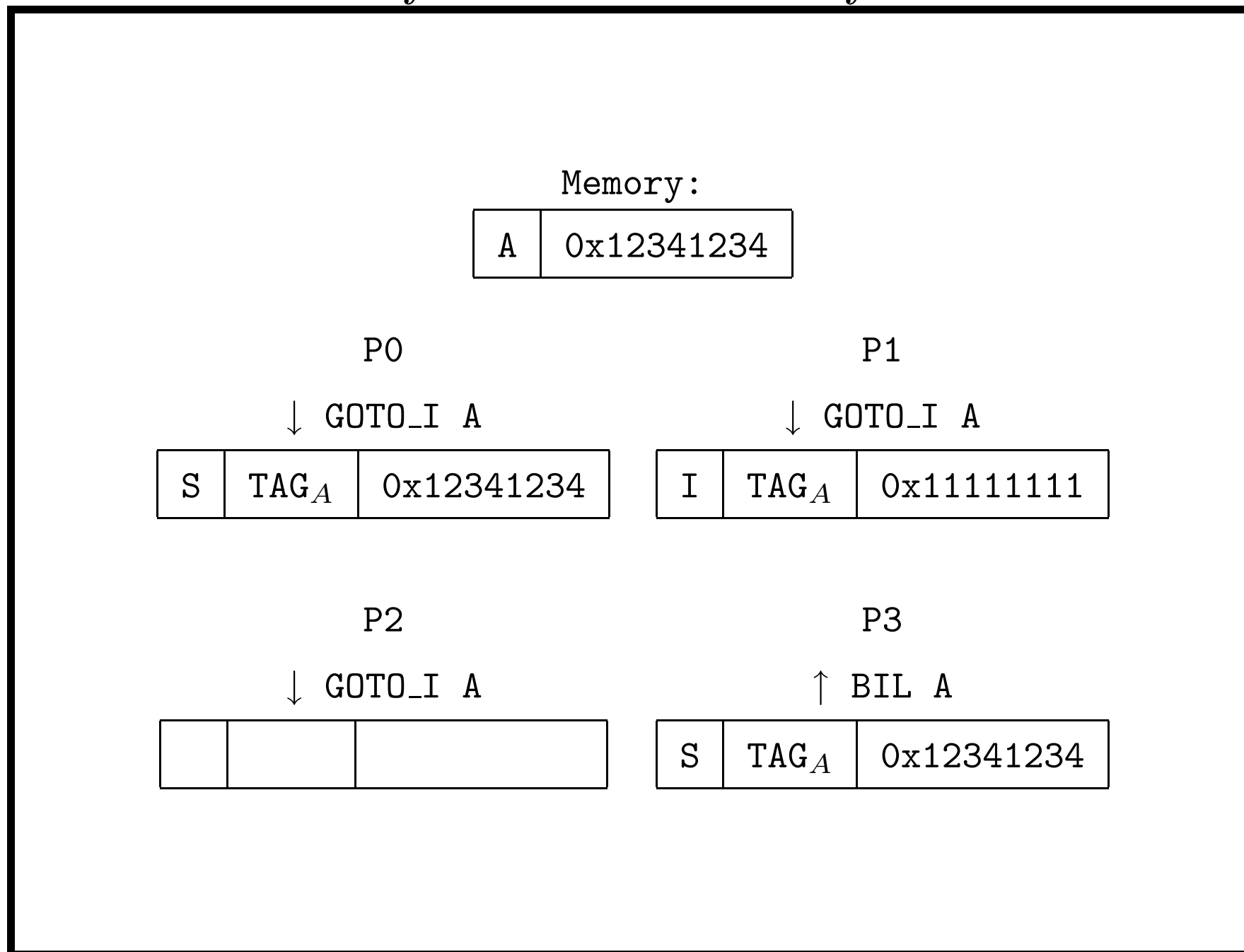


# Atomicity and Cache Coherency For x86



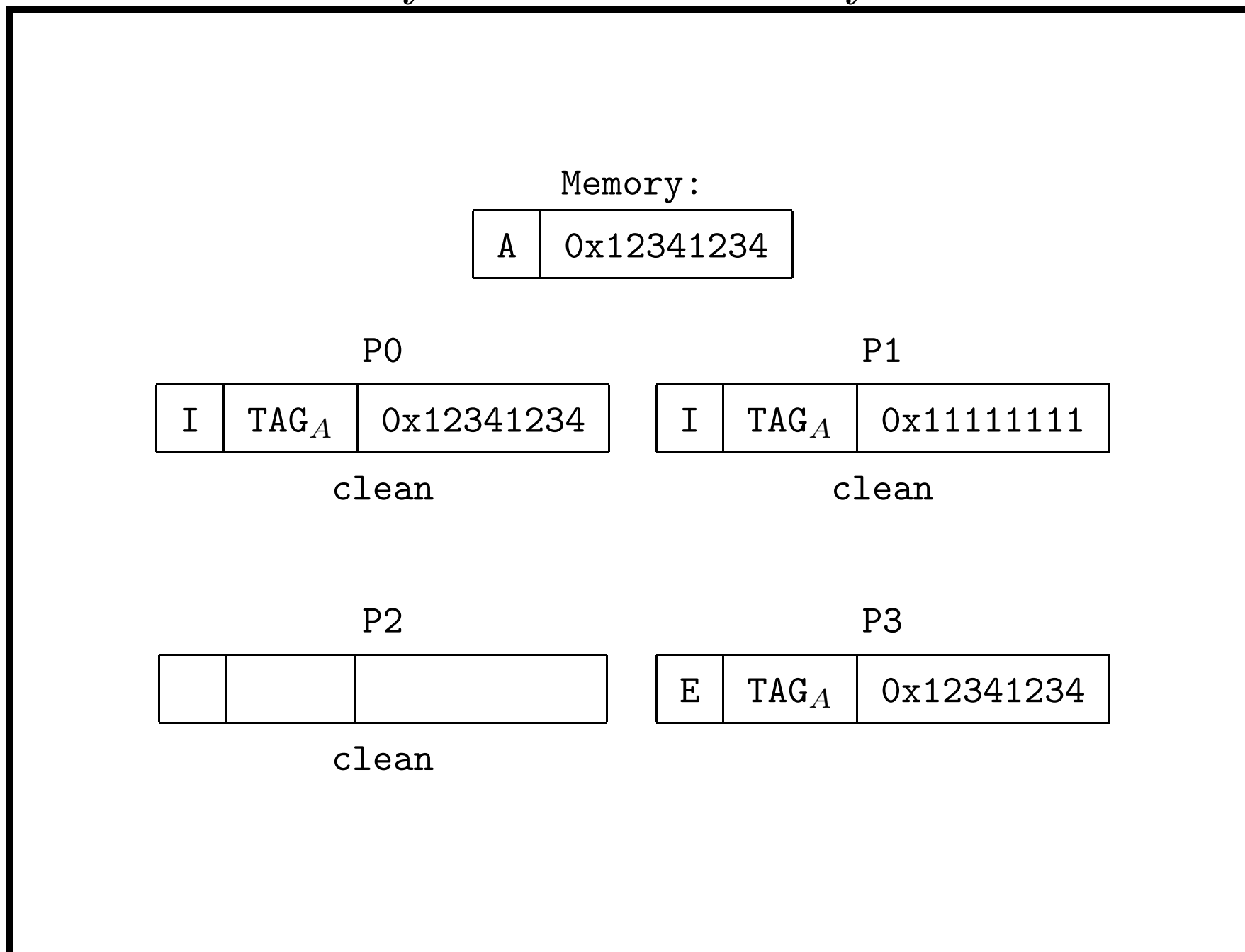


# Atomicity and Cache Coherency For x86



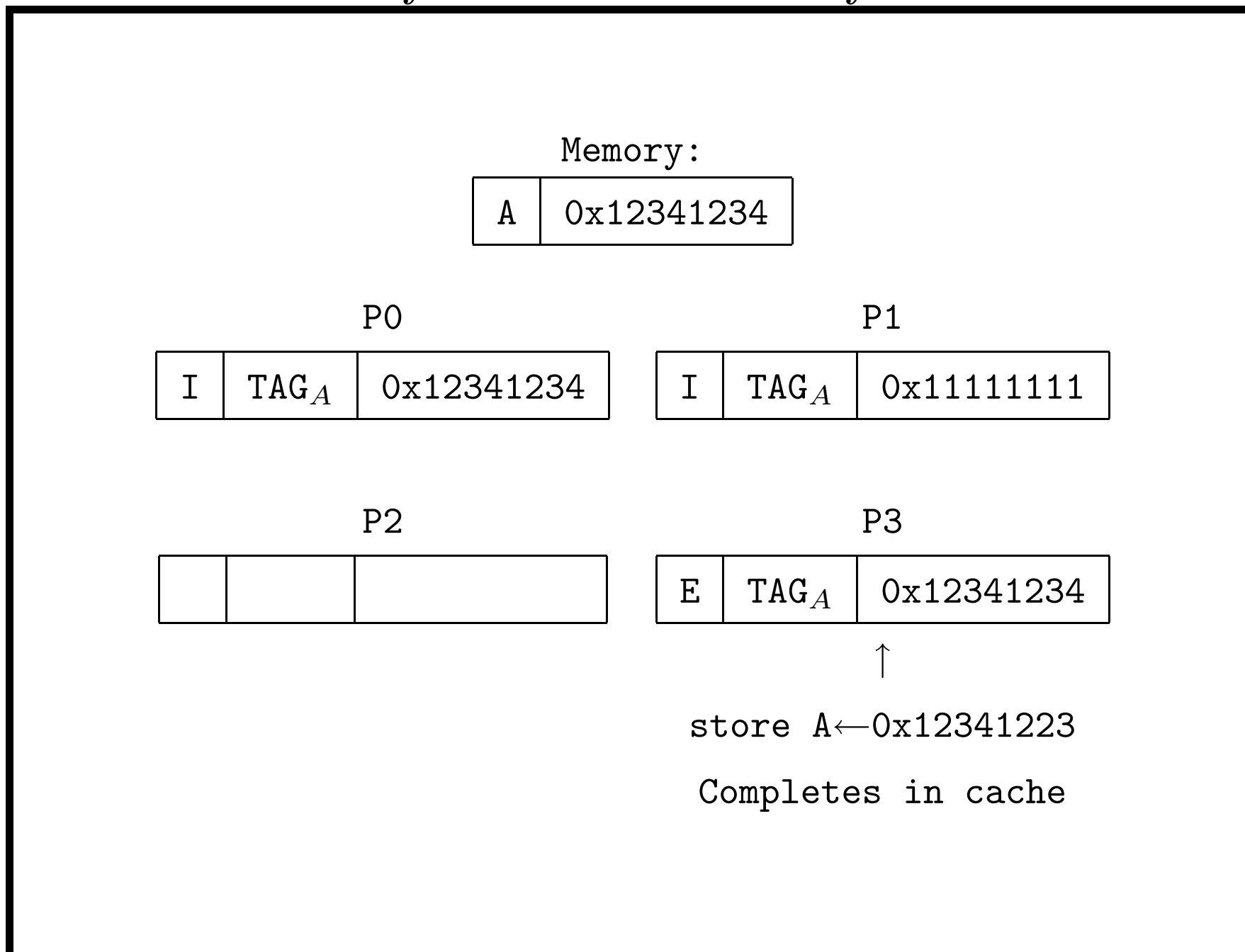


# Atomicity and Cache Coherency For x86



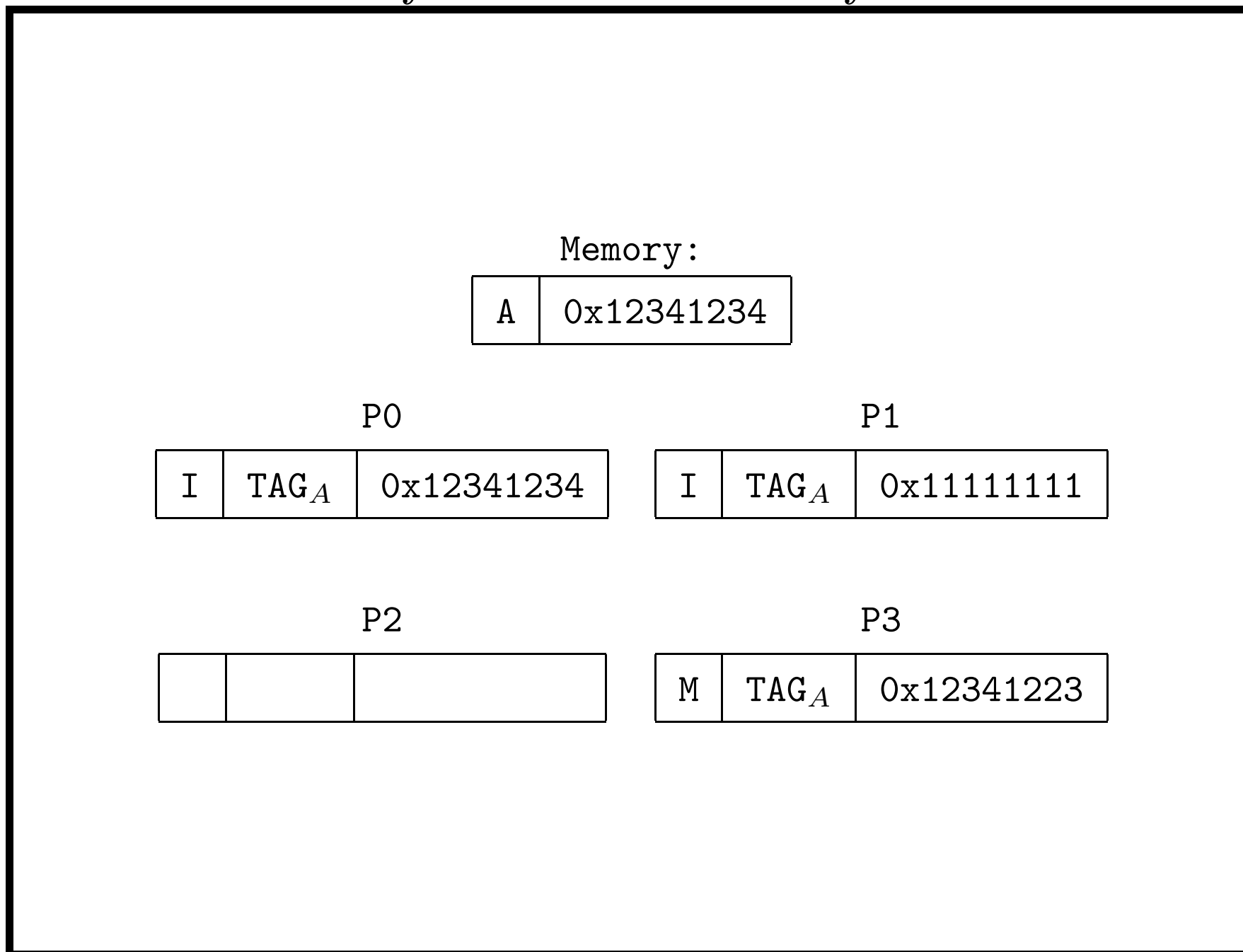


# Atomicity and Cache Coherency For x86



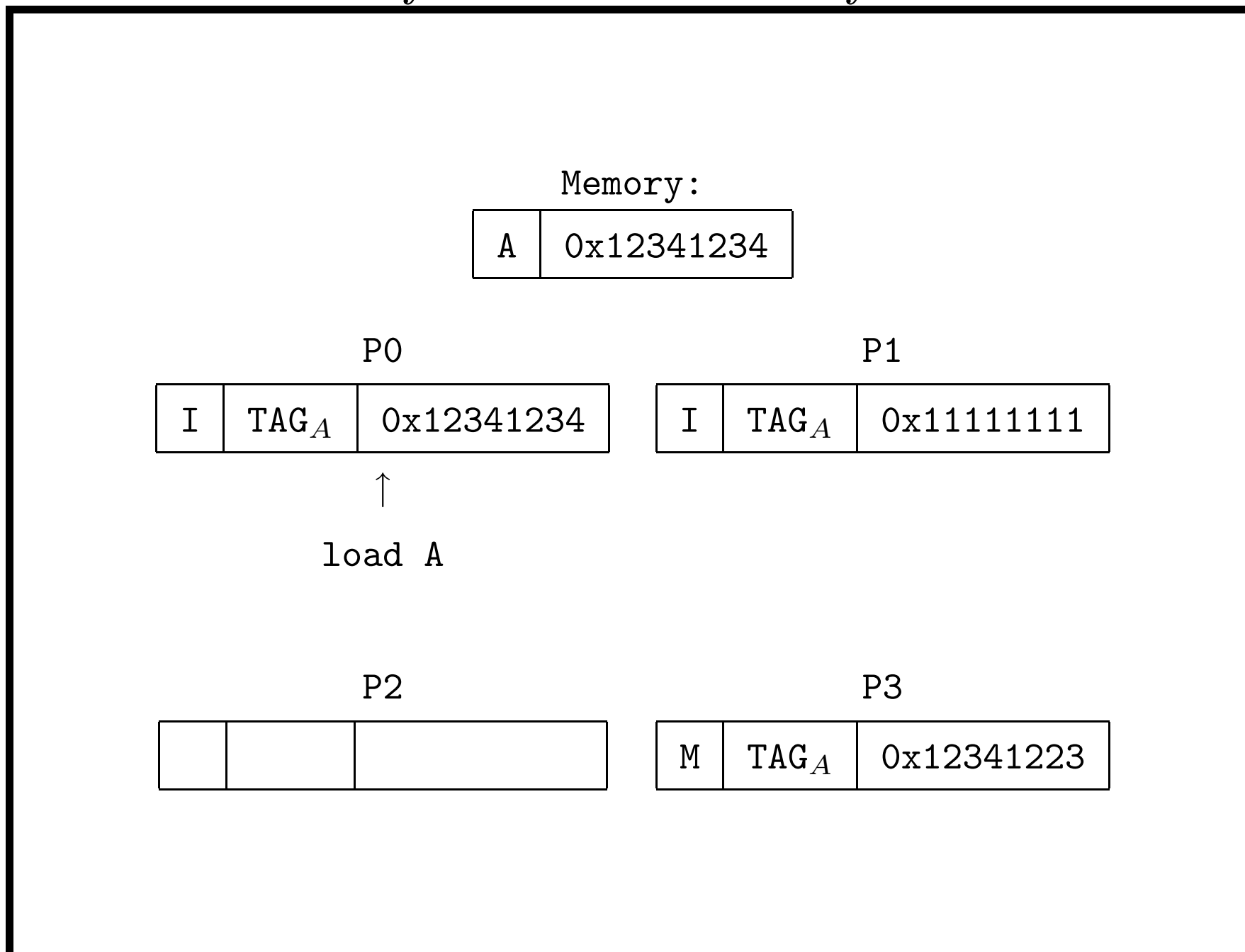


# Atomicity and Cache Coherency For x86



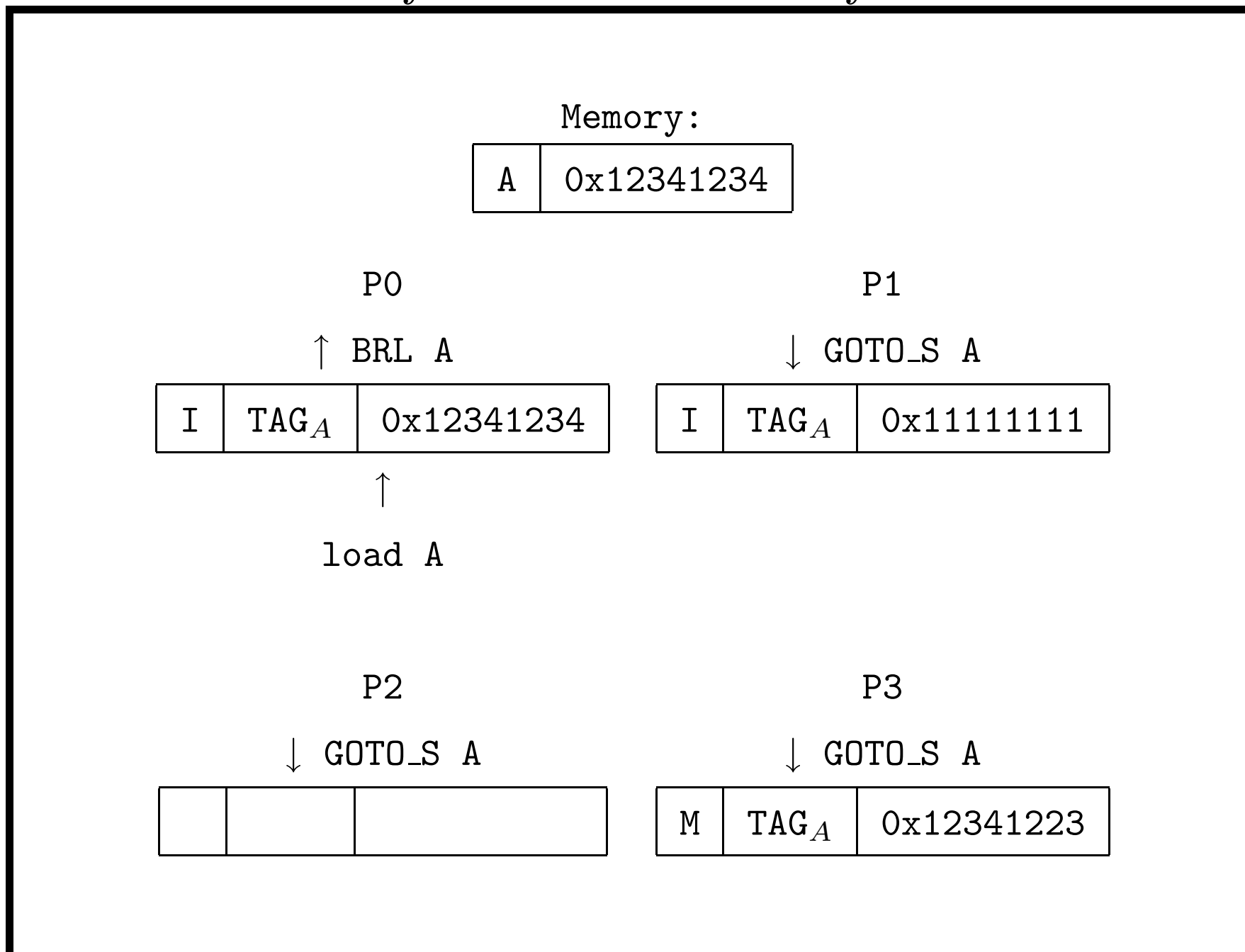


# Atomicity and Cache Coherency For x86



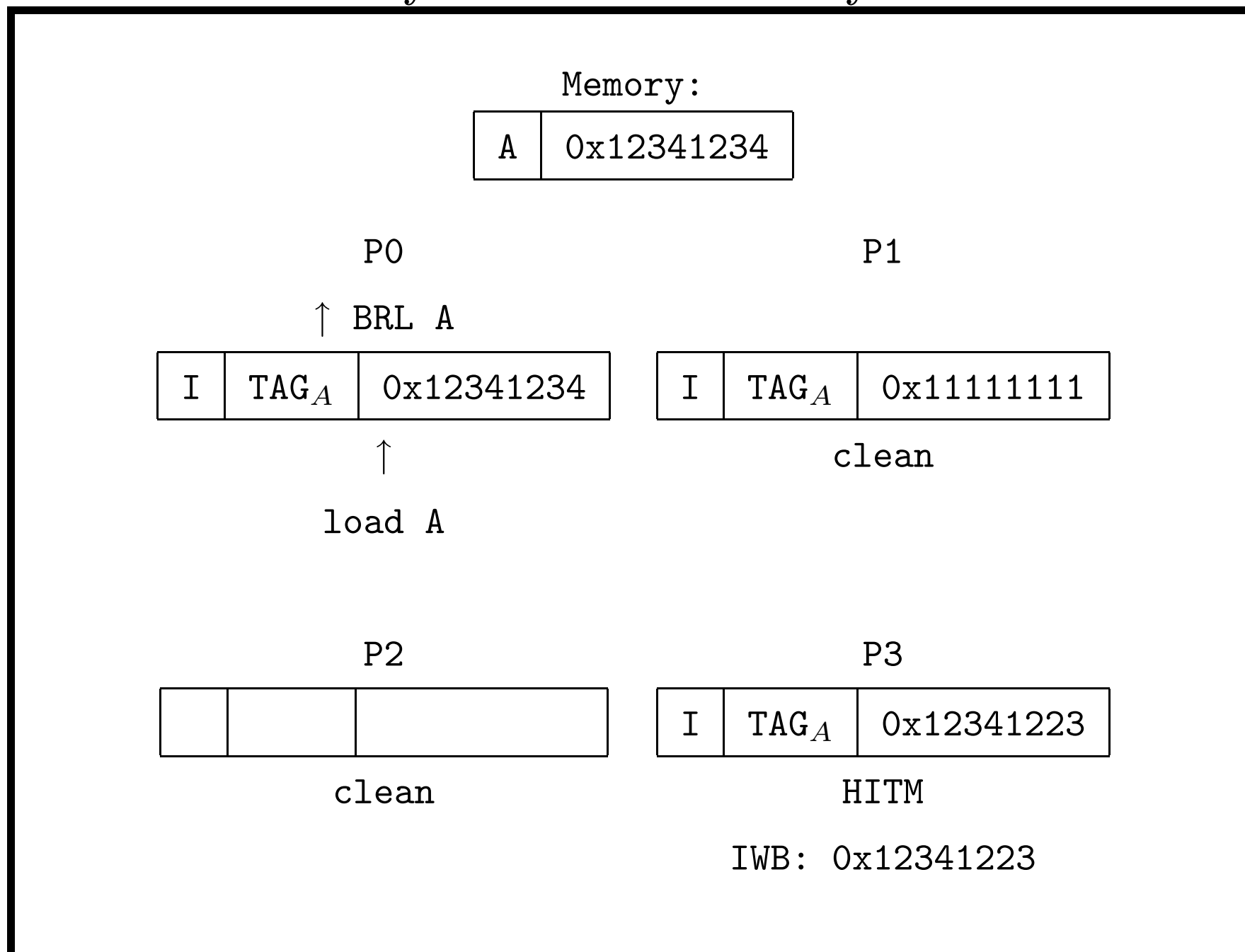


# Atomicity and Cache Coherency For x86



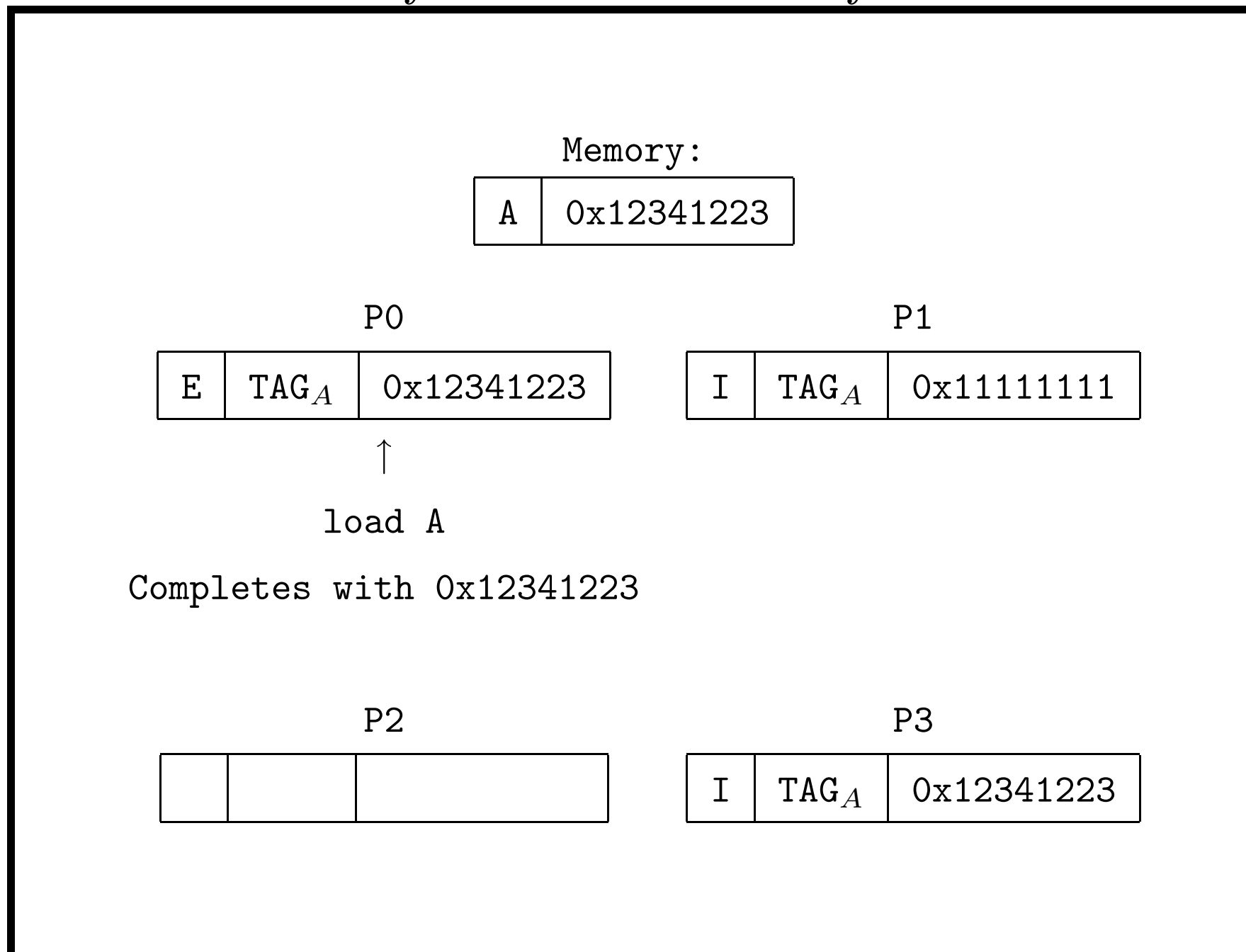


# Atomicity and Cache Coherency For x86



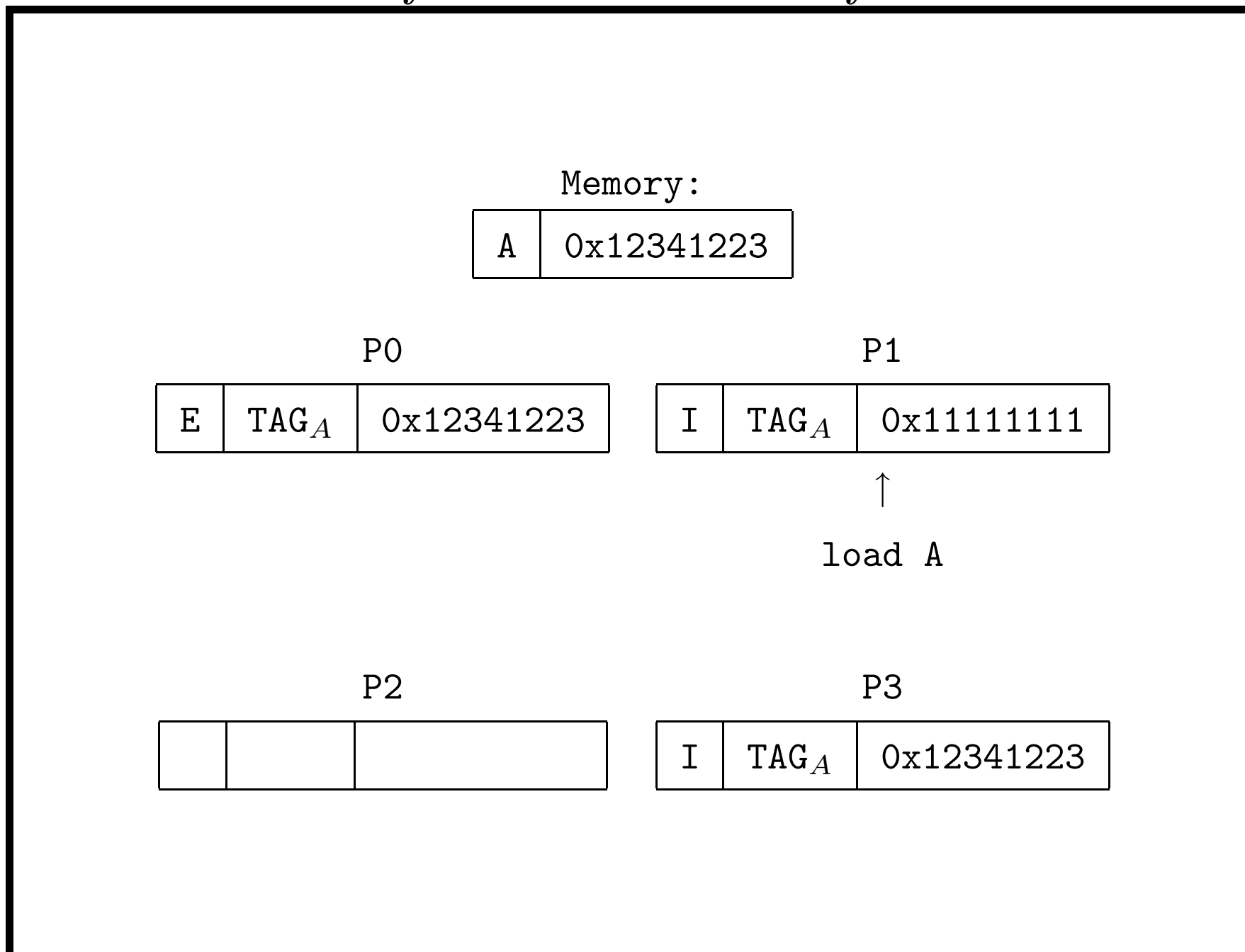


# Atomicity and Cache Coherency For x86



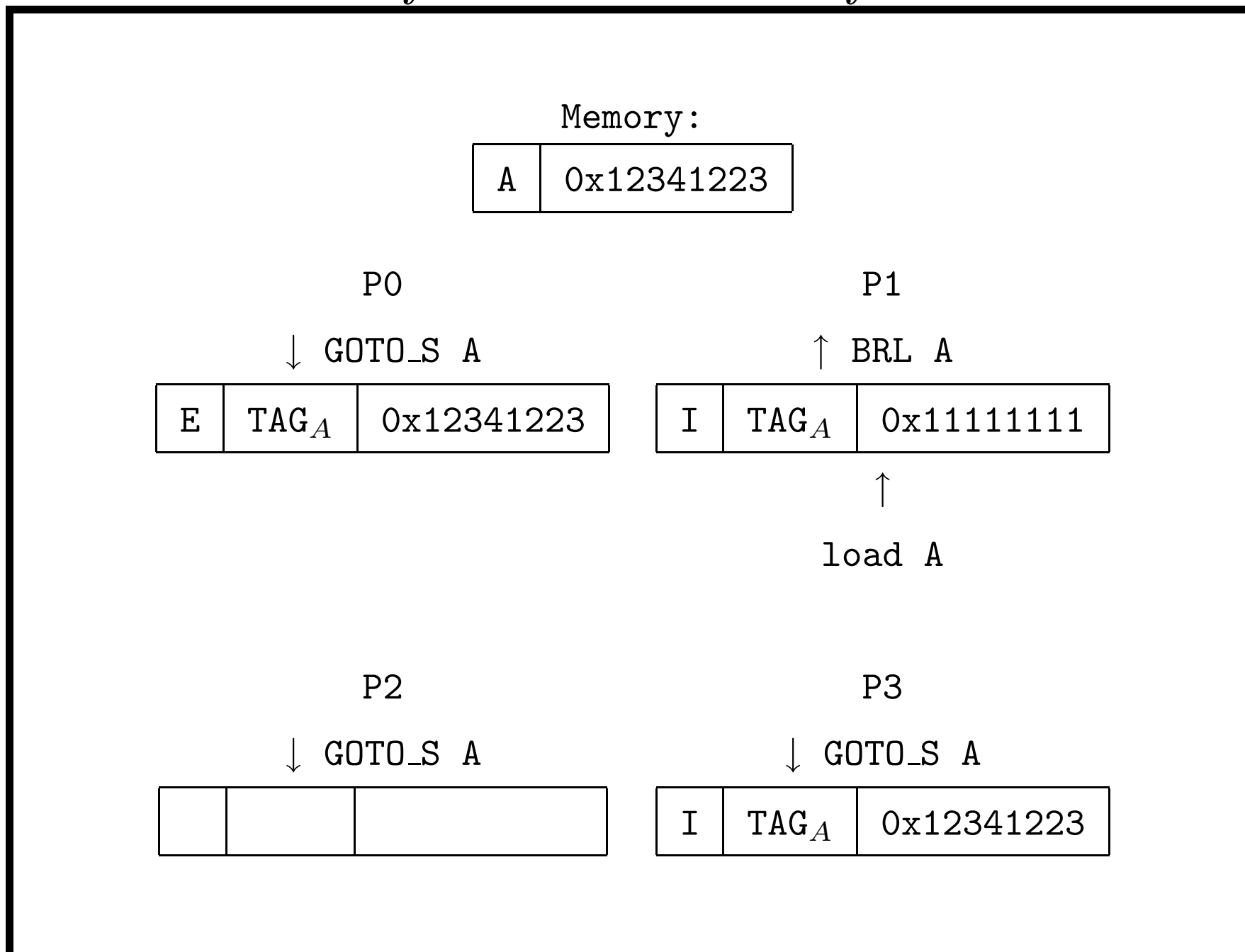


# Atomicity and Cache Coherency For x86



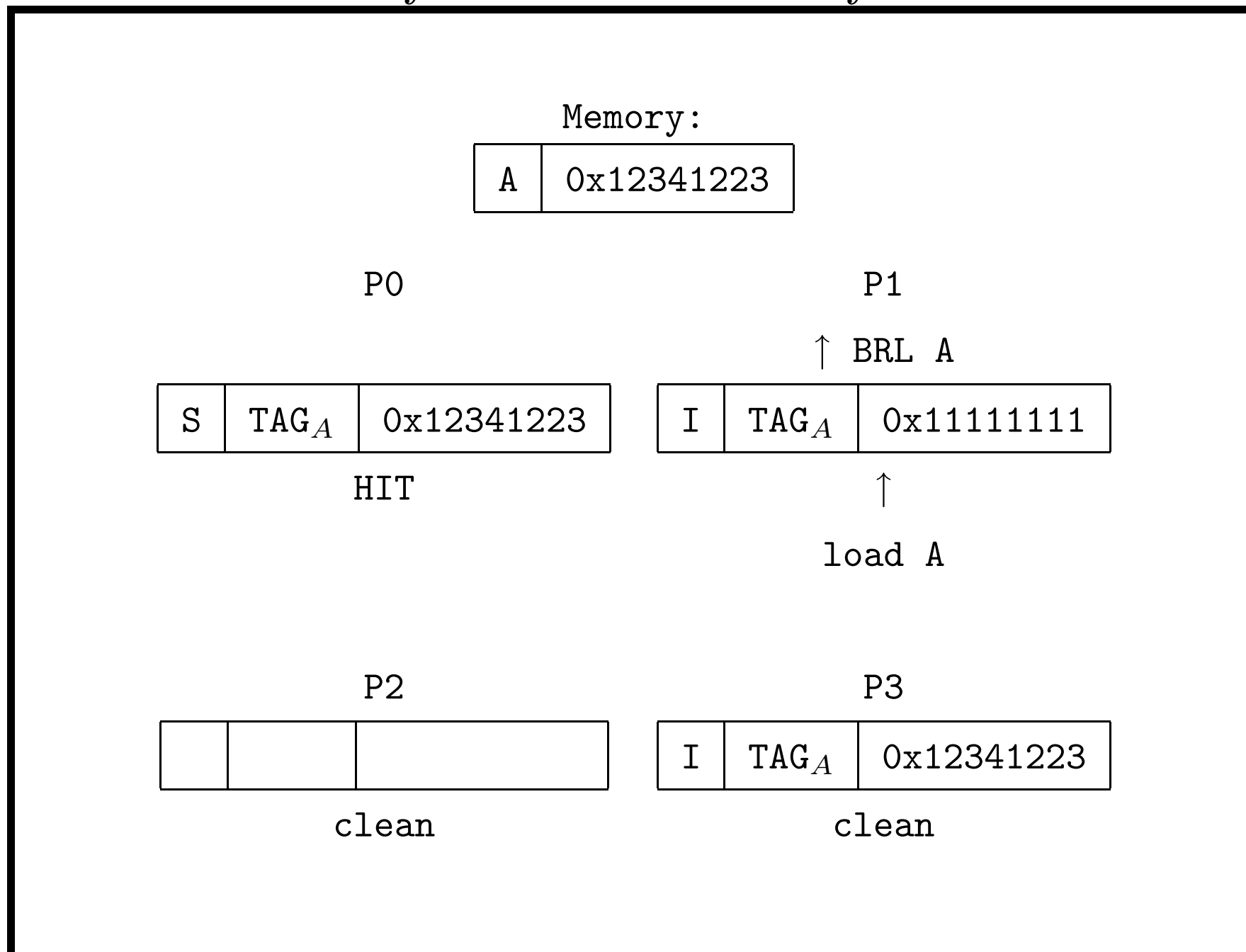


# Atomicity and Cache Coherency For x86



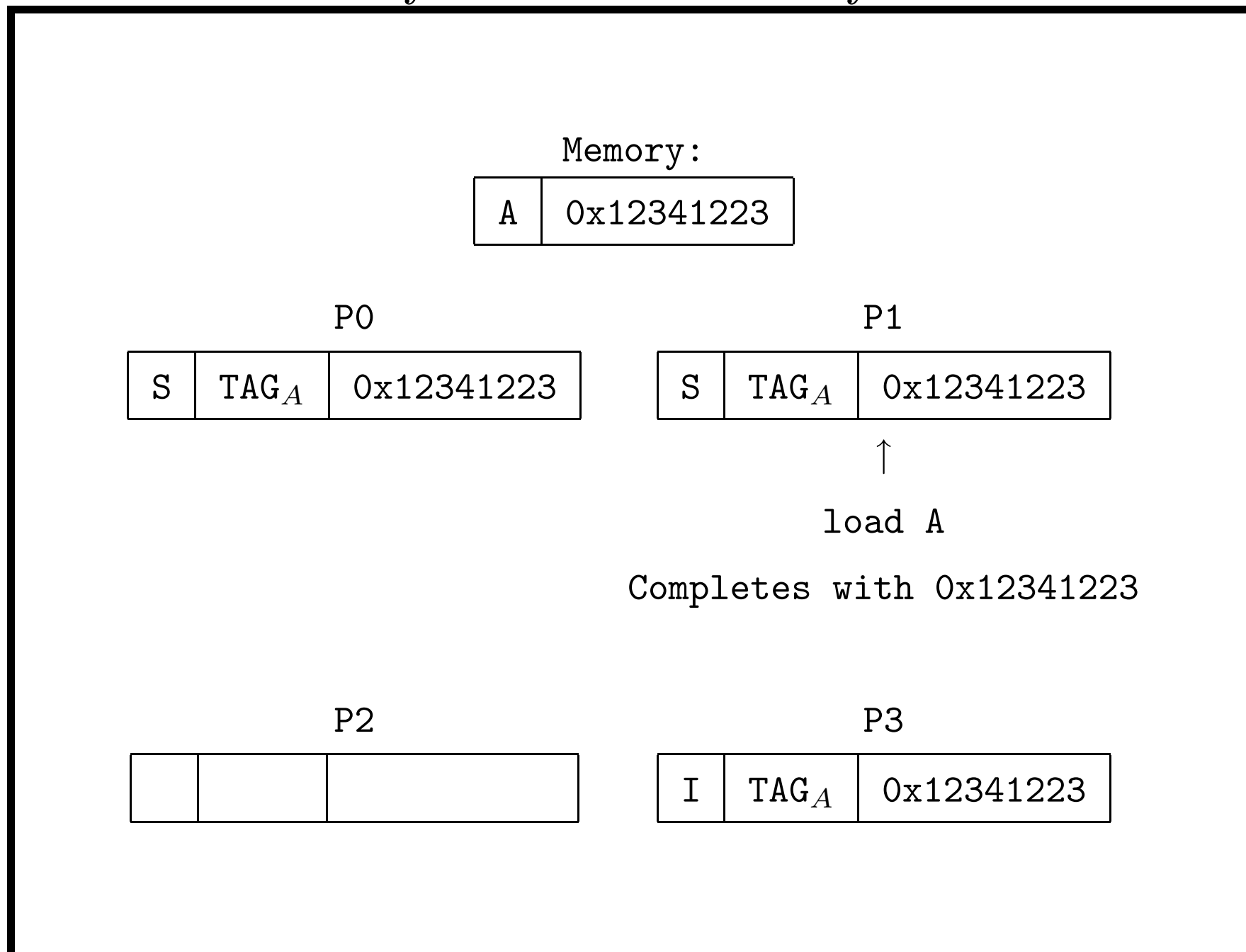


# Atomicity and Cache Coherency For x86



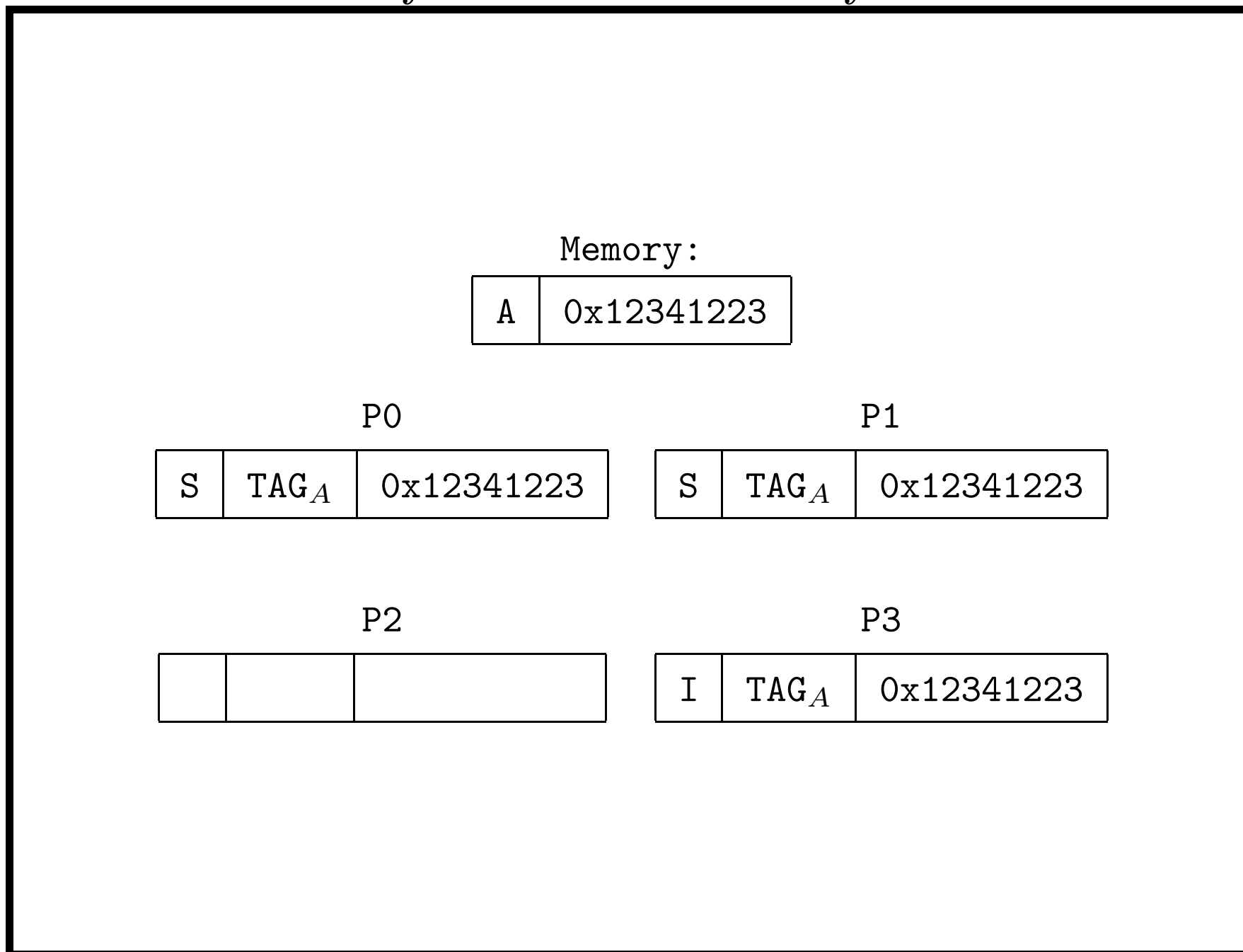


# Atomicity and Cache Coherency For x86





# Atomicity and Cache Coherency For x86





## Atomicity and Cache Coherency For x86

Questions on x86 Cache  
Coherency?



## Atomicity and Cache Coherency For x86

### Atomicity

Memory request or set of related requests occurring as an indivisible operation

- Request Atomicity
- Read-Modify-Write Atomicity



## Atomicity and Cache Coherency For x86

### Request Atomicity

Initially  $A = 0x0000$

P0

$A \leftarrow 0x1122$

P1

$A \leftarrow 0x3344$

P2

load A

What 16-bit value does P2 load?



## Atomicity and Cache Coherency For x86

### Request Atomicity

- If requests are atomic: 0x0000, 0x1122, or 0x3344
- Atomicity is not guaranteed if A is misaligned
- If A is misaligned, request may be split

$$A \leftarrow 0x3344 \left\{ \begin{array}{l} \text{store\_8 } A \leftarrow 0x44 \\ \text{store\_8 } A+1 \leftarrow 0x33 \end{array} \right.$$

# Atomicity and Cache Coherency For x86



## The Frankenstein Load

P0	P1	P2
A ← 0x1122	A ← 0x3344	load A
<hr/>		
	store_8 A ← 0x44	
		load_8 A
store_8 A ← 0x22		
store_8 A+1 ← 0x11		
		load_8 A+1
	store_8 A+1 ← 0x33	
<hr/>		

P2 loads 0x1144, a value which was never written!



## Atomicity and Cache Coherency For x86

### Request Atomicity

- Bus protocol doesn't allow transaction to span lines
- Transaction which is split is not atomic
- Aligned request guaranteed to be atomic
- **Never use misaligned data, especially in SMP system**
- Usually taken care of by compiler



## Atomicity and Cache Coherency For x86

### Read-Modify-Write Atomicity

Initially  $A = 0$

P0	P1	P2
ADD A, 1	ADD A, 1	ADD A, 1

What value does A contain after all processors execute?



## Read-Modify-Write Atomicity

Add is a read-modify-write instruction

ADD A, 1 {  
load\_with\_store\_intent tmp←A  
add tmp, 1  
store A←tmp  
set\_flags

Other transactions may occur between load and store



## Atomicity and Cache Coherency For x86

### Read-Modify-Write Atomicity

P0	P1	P2
ADD A, 1	ADD A, 1	ADD A, 1
<hr/>		
LWSI A (0)		
	LWSI A (0)	
	store A (1)	
		LWSI A (1)
store A (1)		
		store A (2)



## Read-Modify-Write Atomicity

LOCK prefix makes read-modify-write instruction atomic

```
LOCK ADD A, 1 {
  load_lock tmp ← A
  add tmp, 1
  store_unlock A ← tmp
  set_flags
}
```

No transactions to A may occur between load and store



## Read-Modify-Write Atomicity

P0	P1	P2
LOCK ADD A, 1	LOCK ADD A, 1	LOCK ADD A, 1
<hr/>		
	load_lock A (0)	
	stunlock A (1)	
load_lock A (1)		
stunlock A (2)		
		load_lock A (2)
		stunlock A (3)



## Atomicity and Cache Coherency For x86

Questions on x86 Atomicity?



## Implementing Synchronization Primitives

- Mutex acquire/release
- Sync barrier
- Semaphore



## Atomicity and Cache Coherency For x86

### Test-and-set Approach

- Very simple, intuitive approach
- `var` is aligned 32-bit value initialized to 0
- **Performance disaster if mutex is contended**

```
mutex_acquire:                mutex_release:
    LOCK  BTS  var, 0          MOV  var, 0
    JC   mutex_acquire
```



### Performance Problem: “Snoop Storm”

- `mutex_acquire` is very tight loop
- Every iteration stores to shared memory location
- Each waiting processor needs `var` in E/M each iteration
- Bus is saturated with `GOTO_I` snoops getting HITM (if more than 1 processor is waiting)
- Available bus bandwidth to mutex owner is greatly reduced



## Atomicity and Cache Coherency For x86

### Test-and-test-and-set Approach

- Most of wait is in top loop with no store
- All waiting processors can have var in S in top loop
- Top loop executes completely in cache
- Substantially reduces snoop traffic on bus

mutex\_acquire:

```
TEST  var, 1
JNZ   mutex_acquire
LOCK  BTS  var, 0
JC    mutex_acquire
```

mutex\_release:

```
MOV  var, 0
```



## Atomicity and Cache Coherency For x86

### Test-and-wait Approach

- Supported by new MONITOR/MWAIT instructions
- Waiting processors enter very lightweight “sleep” state
- Processors wake on the mutex\_release store



## Atomicity and Cache Coherency For x86

```
mutex_acquire:                                mutex_release:
    LOCK  BTS  var, 0                          MOV  var, 0
    JNC   mutex_owned
    LEA  EAX, var
    XOR  ECX, ECX
    XOR  EDX, EDX
    MONITOR
    XOR  EAX, EAX
    TEST var, 1
    JZ   mutex_acquire
    MWAIT
    JMP  mutex_acquire
mutex_owned:
```



## Atomicity and Cache Coherency For x86

### Barriers and Semaphores

- For semaphores LOCK XADD instruction works nicely
- Should use with CMP for test-and-test-and-set approach
- Barriers inherently have read-only spin loop

```
barrier:
```

```
    LOCK SUB var, 1
```

```
barrier_spin:
```

```
    CMP var, 0
```

```
    JNE barrier_spin
```



## Hyper-threading Performance Implications

- Core resources divided between threads  
(some statically, some dynamically)
- Spin loop executes extremely efficiently
- Waiting thread will densely fill pipeline
- **Test-and-wait forces waiting thread to yield resources**



## Hyper-threading and the Pause Instruction

- P4 systems are deeply speculative
- Many iterations of the spin loop in flight when store occurs
- Memory pipeline must flush to avoid ordering violation
- Pause instruction blocks speculation
- Avoids “nuke” and frees resources for other thread



## Atomicity and Cache Coherency For x86

### Test-and-test-and-set With Pause

```
mutex_acquire:                mutex_release:
    PAUSE                      MOV  var, 0
    TEST  var, 1
    JNZ   mutex_acquire
    LOCK  BTS  var, 0
    JC    mutex_acquire
```

Only 1 iteration of loop in flight at a time